1. topic.

# TODO

SELinux

06-01 00:46:47.950 222-222/? E/SELinux: avc: denied { find } for service=v1 pid=7448 uid=10049 scontext=u:r:platform\_app:s0:c512,c768 tcontext=u:object\_r:default\_android\_service:s0 tclass=service\_manager permissive=1

<https://blog.csdn.net/innost/article/details/19299937/>

<https://blog.csdn.net/csh86277516/article/details/78578836>

Android SELinux avc dennied权限问题

1. App和系统签名的原理是什么？

Android 安全设计

Android 安全框架 -- 总概http://blog.csdn.net/blue\_rush/article/details/55045546

<https://security.tencent.com/index.php/blog/msg/38>

# UserHandle

Fgh

## getAppId

/\*\*

\* @hide Range of uids allocated for a user.

\*/

public static final int PER\_USER\_RANGE = 100000;

@TestApi

public static @AppIdInt int getAppId(int uid) {

return uid % PER\_USER\_RANGE;

}

## isSameApp

public static boolean isSameApp(int uid1, int uid2) {

return getAppId(uid1) == getAppId(uid2);

}

## getUserId

/\*\*

\* Returns the user id for a given uid.

\* @hide

\*/

public static @UserIdInt int getUserId(int uid) {

if (MU\_ENABLED) {

return uid / PER\_USER\_RANGE;

} else {

return UserHandle.USER\_SYSTEM;

}

}

## getCallingUserId

/\*\* @hide \*/

public static @UserIdInt int getCallingUserId() {

return getUserId(Binder.getCallingUid());

}

## isIsolated

孤立的；隔离的；分离的

/\*\* @hide \*/

public static boolean isIsolated(int uid) {

if (uid > 0) {

final int appId = getAppId(uid);

return appId >= Process.FIRST\_ISOLATED\_UID && appId <= Process.LAST\_ISOLATED\_UID;

} else {

return false;

}

}

/\*\* @hide \*/

public static boolean isApp(int uid) {

if (uid > 0) {

final int appId = getAppId(uid);

return appId >= Process.FIRST\_APPLICATION\_UID && appId <= Process.LAST\_APPLICATION\_UID;

} else {

return false;

}

}

# Anroid 6 动态权限申请

对于6.0以下的权限及在安装的时候，根据权限声明产生一个权限列表，用户只有在同意之后才能完成app的安装，造成了我们想要使用某个app，就要默默忍受其一些不必要的权限（比如是个app都要访问通讯录、短信等）。而在6.0以后，我们可以直接安装，当app需要我们授予不恰当的权限的时候，我们可以予以拒绝（比如：单机的象棋对战，请求访问任何权限，我都是不同意的）。当然你也可以在设置界面对每个app的权限进行查看，以及对单个权限进行授权或者解除授权。

新的权限机制更好的保护了用户的隐私，Google将权限分为两类，一类是Normal Permissions，这类权限一般不涉及用户隐私，是不需要用户进行授权的，比如手机震动、访问网络等；另一类是Dangerous Permission，一般是涉及到用户隐私的，需要用户进行授权，比如读取sdcard、访问通讯录等。

## 权限分类

### Normal Permissions如下

ACCESS\_LOCATION\_EXTRA\_COMMANDS

ACCESS\_NETWORK\_STATE

ACCESS\_NOTIFICATION\_POLICY

ACCESS\_WIFI\_STATE

BLUETOOTH

BLUETOOTH\_ADMIN

BROADCAST\_STICKY

CHANGE\_NETWORK\_STATE

CHANGE\_WIFI\_MULTICAST\_STATE

CHANGE\_WIFI\_STATE

DISABLE\_KEYGUARD

EXPAND\_STATUS\_BAR

GET\_PACKAGE\_SIZE

INSTALL\_SHORTCUT

INTERNET

KILL\_BACKGROUND\_PROCESSES

MODIFY\_AUDIO\_SETTINGS

NFC

READ\_SYNC\_SETTINGS

READ\_SYNC\_STATS

RECEIVE\_BOOT\_COMPLETED

REORDER\_TASKS

REQUEST\_INSTALL\_PACKAGES

SET\_ALARM

SET\_TIME\_ZONE

SET\_WALLPAPER

SET\_WALLPAPER\_HINTS

TRANSMIT\_IR

UNINSTALL\_SHORTCUT

USE\_FINGERPRINT

VIBRATE

WAKE\_LOCK

WRITE\_SYNC\_SETTINGS

### Dangerous Permissions

group:android.permission-group.CONTACTS

permission:android.permission.WRITE\_CONTACTS

permission:android.permission.GET\_ACCOUNTS

permission:android.permission.READ\_CONTACTS

group:android.permission-group.PHONE

permission:android.permission.READ\_CALL\_LOG

permission:android.permission.READ\_PHONE\_STATE

permission:android.permission.CALL\_PHONE

permission:android.permission.WRITE\_CALL\_LOG

permission:android.permission.USE\_SIP

permission:android.permission.PROCESS\_OUTGOING\_CALLS

permission:com.android.voicemail.permission.ADD\_VOICEMAIL

group:android.permission-group.CALENDAR

permission:android.permission.READ\_CALENDAR

permission:android.permission.WRITE\_CALENDAR

group:android.permission-group.CAMERA

permission:android.permission.CAMERA

group:android.permission-group.SENSORS

permission:android.permission.BODY\_SENSORS

group:android.permission-group.LOCATION

permission:android.permission.ACCESS\_FINE\_LOCATION

permission:android.permission.ACCESS\_COARSE\_LOCATION

group:android.permission-group.STORAGE

permission:android.permission.READ\_EXTERNAL\_STORAGE

permission:android.permission.WRITE\_EXTERNAL\_STORAGE

group:android.permission-group.MICROPHONE

permission:android.permission.RECORD\_AUDIO

group:android.permission-group.SMS

permission:android.permission.READ\_SMS

permission:android.permission.RECEIVE\_WAP\_PUSH

permission:android.permission.RECEIVE\_MMS

permission:android.permission.RECEIVE\_SMS

permission:android.permission.SEND\_SMS

permission:android.permission.READ\_CELL\_BROADCASTS

可以通过adb shell pm list permissions -d -g进行查看。

看到上面的dangerous permissions，会发现一个问题，好像危险权限都是一组一组的，恩，没错，的确是这样的，

那么有个问题：分组对我们的权限机制有什么影响吗？

的确是有影响的，如果app运行在Android 6.x的机器上，对于授权机制是这样的。如果你申请某个危险的权限，假设你的app早已被用户授权了**同一组**的某个危险权限，那么系统会立即授权，而不需要用户去点击授权。比如你的app对READ\_CONTACTS已经授权了，当你的app申请WRITE\_CONTACTS时，系统会直接授权通过。此外，对于申请时弹出的dialog上面的文本说明也是对整个权限组的说明，而不是单个权限（ps:这个dialog是不能进行定制的）。

不过需要注意的是，不要对权限组过多的依赖，尽可能对每个危险权限都进行正常流程的申请，因为在后期的版本中这个权限组可能会产生变化。

## API

### 检查权限hasPermissions

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **private** **boolean** hasPermissions() {
2. PackageManager pm = getPackageManager();
3. String packageName = getPackageName();
4. **int** granted = (mAudioToggle.isChecked() ? pm.checkPermission(RECORD\_AUDIO, packageName) : PackageManager
5. .PERMISSION\_GRANTED)
6. | pm.checkPermission(WRITE\_EXTERNAL\_STORAGE, packageName);
7. **return** granted == PackageManager.PERMISSION\_GRANTED;
8. }

### 请求权限requestPermissions

g

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. @TargetApi(M)
2. **private** **void** requestPermissions() {
3. String[] permissions = mAudioToggle.isChecked()
4. ? **new** String[]{WRITE\_EXTERNAL\_STORAGE, RECORD\_AUDIO}
5. : **new** String[]{WRITE\_EXTERNAL\_STORAGE};
6. **boolean** showRationale = **false**;
7. **for** (String perm : permissions) {
8. showRationale |= shouldShowRequestPermissionRationale(perm);
9. }
10. **if** (!showRationale) {
11. requestPermissions(permissions, REQUEST\_PERMISSIONS);
12. **return**;
13. }

### 回调结果onRequestPermissionsResult

Sdf

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. @Override
2. **public** **void** onRequestPermissionsResult(**int** requestCode, String[] permissions, **int**[] grantResults) {
3. **if** (requestCode == REQUEST\_PERMISSIONS) {
4. **int** granted = PackageManager.PERMISSION\_GRANTED;
5. **for** (**int** r : grantResults) {
6. granted |= r;
7. }
8. **if** (granted == PackageManager.PERMISSION\_GRANTED) {
9. startCaptureIntent();
10. } **else** {
11. toast("No Permission!");
12. }
13. }
14. }

## REF

[在运行时请求权限](https://developer.android.com/training/permissions/requesting?hl=zh-cn)

## checkPermission原理

### app：ActivityManagerNative

public int checkPermission(String permission, int pid, int uid)

throws RemoteException {

Parcel data = Parcel.obtain();

Parcel reply = Parcel.obtain();

data.writeInterfaceToken(IActivityManager.descriptor);

data.writeString(permission);

data.writeInt(pid);

data.writeInt(uid);

mRemote.transact(CHECK\_PERMISSION\_TRANSACTION, data, reply, 0);

reply.readException();

int res = reply.readInt();

data.recycle();

reply.recycle();

return res;

}

### ActMS. checkPermission

public int checkPermission(String permission, int pid, int uid) {

if (permission == null) {

return PackageManager.PERMISSION\_DENIED;

}

return checkComponentPermission(permission, pid, UserHandle.getAppId(uid), -1, true);

}

### ActMS .checkComponentPermission

int checkComponentPermission(String permission, int pid, int uid,

int owningUid, boolean exported) {

if (pid == MY\_PID) {

return PackageManager.PERMISSION\_GRANTED;

}

return ActivityManager.checkComponentPermission(permission, uid,

owningUid, exported);

}

### ActivityManager.checkComponentPermission

Dsf

/\*\* @hide \*/

public static int checkComponentPermission(String permission, int uid,

int owningUid, boolean exported) {

// Root, system server get to do everything.

<!--root及System能获取所有权限-->

if (uid == Process.ROOT\_UID || uid == Process.SYSTEM\_UID) {

return PackageManager.PERMISSION\_GRANTED;

}

。。。

<!--普通的权限查询-->

try {

return AppGlobals.getPackageManager()

.checkUidPermission(permission, uid);

} catch (RemoteException e) {

// Should never happen, but if it does... deny!

Slog.e(TAG, "PackageManager is dead?!?", e);

}

return PackageManager.PERMISSION\_DENIED;

}

### PkgMS.checkUidPermission

最终调用PackageManagerService.java去查看是否有权限，到这里，我们只需要知道权限的查询其实是通过PKMS来进行的。心里先有个底，权限的更新，持久化，恢复都是通过PKMS来进行的。

### PMS

<https://www.jianshu.com/p/9938d367b6db>

# SEAndroid

## 概述

### SEAndroid背景

SEAndroid(Security Enhanced Android)是美国国家安全局（NSA）根据Android 以Linux 为内核基础的特性，将SELinux 修改移植到Android 系统上而形成的。SELinux通过提前编写各个进程的安全操作规则，防止其进行超越权限的访问来达到保障安全的目的。

SEAndroid 一开始的关注点在于将SELinux 的机制引入到Android 系统中，但是这仅仅是解决了底层的权限管理问题，考虑到这种问题，SEAndroid 结合了Android 的特征，在Framework 层参照了SELinux 的权限管理模式，实现了一种称为Middleware MAC（简称为MAC）的强制访问控制方式。

其安全结构可以简单表述为：SEAndroid =Android+MAC+ SELinux

因为SEAndroid 目前正处于发展阶段，在master 源码库里面只存在Install-time MAC 的强制访问控制方式。在分支代码库里面存在其它类型的强制访问控制方式。

### SEAndroid的安全机制

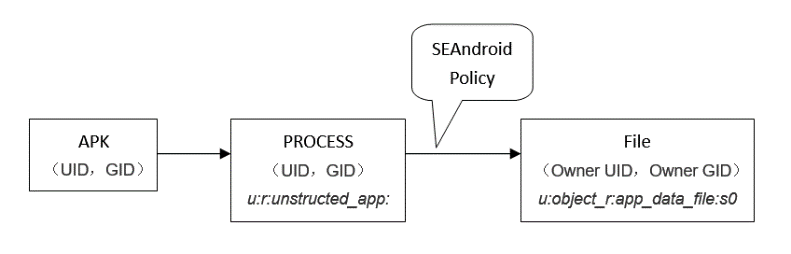


图1. SEAndroid安全机制

SEAndroid是一种基于安全策略的MAC安全机制，这种安全策略实施在主体和客体的安全上下文之上（主体一般就是进程，而客体一般就是文件）。这个安全上下文由用户、角色、类型、安全级别四个部分组成，每一部分通过一个冒号来分隔。

例如，图1斜体部分即为SEAndroid安全上下文。其中，u:r:unstructed\_app:s0描述的是用户安装的APK所在进程的安全上下文，而u:object\_r:app\_data\_file:s0描述的是用户安装的APK在运行过程中生成的数据文件的安全上下文。当每一个进程和文件都关联上一个安全上下文之后，系统管理员就可以基于这些安全上下文制定一个安全访问策略，用来规定什么样的进程可以访问什么样的文件。

SEAndroid安全机制与传统的Linux UID/GID安全机制是并存关系的，也就是说，它们同时用来约束进程的权限。当一个进程访问一个文件的时候，首先要通过基于UID/GID的DAC安全检查，接着才有资格进入到基于SEAndroid的MAC安全检查。只要其中的一个检查不通过，那么进程访问文件的请求就会被拒绝。

## SEAndroid架构

以SELinux文件系统接口为边界，SEAndroid安全机制包含有内核空间和用户空间两部分支持。这些内核空间模块和用户空间模块的作用以及交互如下表所示：

1. 用户空间
2. 主要模块
3. 安全上下文（Security Context）
4. 安全服务（Security Server）
5. 安全策略（SEAndroid Policy）
6. 内核空间
7. 主要模块
8. Linux安全模块（SELinux LSM）（通常包括访问向量缓冲（Access Vector Cache）和一个安全服务（Security Server））

### 3.1 内核空间

SELinux安全访问流程：

首先，SELinux会在LSM中注册相应的回调函数。其次，LSM会在相应的内核对象子系统中会加入一些Hook代码，以便SELinux可以进行安全检查 。

SELinux在进行安全检查的时候，首先是看一下自己的Access Vector Cache是否已经有结果。如果有的话，就直接将结果返回给相应的内核子系统就可以了。如果没有的话，就需要到Security Server中去进行检查。检查出来的结果在返回给相应的内核子系统的同时，也会保存在自己的Access Vector Cache中，以便下次可以快速地得到检查结果。

内核中的资源在访问的过程中，一般需要获得三次检查通过：

1. SELinux检查，即基于安全上下文和安全策略的安全检查
2. DAC检查，即基于Linux UID/GID的安全检查
3. 一般性错误检查，例如访问的对象是否存在、访问参数是否正确等

### 用户空间

#### 安全上下文

如前所述，这个标签实际上就是一个字符串，它由四部分内容组成，分别是SELinux用户、SELinux角色、类型、安全级别，每一个部分都通过一个冒号来分隔，格式为”user:role:type:sensitivity”。可通过ps -Z命令查看进程的安全上下文，通过ls -Z来查看文件的安全上下文

下面是针对安全级别和安全类型进行进一步研究分析:

##### 安全级别

它是建立在TE的基础之上的。安全级别最开始的目的是用来对政府分类文件进行访问控制的。在基于安全级别的MAC安全机制中，主体（subject）和客体（object）都关联有一个安全级别。其中，安全级别较高的主体可以读取安全级别较低的客体，而安全级别较低的主体可以写入安全级别较高的客体。前者称为”read down”，而后者称为”write up”。通过这种规则，可以允许数据从安全级别较低的主体流向安全级别较高的主体，而限制数据从安全级别较高的主体流向安全级别较低的主体，从而有效地保护了数据。注意，如果主体和客体的安全级别是相同的，那么主体是可以对客体进行读和写的。

安全级别是由敏感性（Sensitivity）和类别（Category）两部分内容组成的，格式为"sensitivity[:category\\_set]"，其中，category\\_set是可选的。例如，假设我们定义有s0、s1两个Sensitivity，以c0、c1、c2三个Category，那么"s0:c0,c1"表示的就是Sensitivity为s0、Category为c0和c1的一个安全级别。

1

##### 安全类型

四种类型的对象的安全上下文，分别是App进程、App数据文件、系统文件和系统属性。这四种类型对象的安全上下文通过四个文件来描述:mac\_permissions.xml、seapp\_contexts、file\_contexts和property\_contexts，它们均位于external/sepolicy目录中。

在SEAndroid中，通常将用来标注文件的安全上下文中的类型称为file\_type，而用来标注进程的安全上下文的类型称为domain，并且每一个用来描述文件安全上下文的类型都将file\_type设置为其属性，每一个用来进程安全上下文的类型都将domain设置为其属性。

如：type init domain; 表示用type语句来将类型domain设置类型init的属性。

.三种文件安全上下文关联方式

1.设置打包在ROM里面的文件的安全上下文

2.设置虚拟文件系统的安全上下文

3.设置应用程序数据文件的安全上下文

.两种进程安全上下文设置方式

1.为独立进程静态地设置安全上下文

2.为应用程序进程设置安全上下文

#### 安全策略

SEAndroid安全机制中的安全策略是在安全上下文的基础上进行描述的，也就是说，它通过主体和客体的安全上下文，定义主体是否有权限访问客体。SEAndroid安全机制主要是使用对象安全上下文中的类型来定义安全策略，这种安全策略就称Type Enforcement，简称TE。在external/sepolicy目录中，所有以.te为后缀的文件经过编译之后，就会生成一个sepolicy文件。这个sepolicy文件会打包在ROM中，并且保存在设备上的根目录下，即它在设备上的路径为/sepolicy。

其核心思想就是最小权限原则，即主体对客体拥有的权限必须要通过allow语句定义才允许，否则的话，一切都是禁止的。

.SEAndroid 的策略文件

SEAndroid 源码中的策略文件位于seandroid/external/sepolicy 中，其用于编译时候生成内核策略库文件，其中有几个重要的文件，其名称及功能如下：

文件名 功能

file\_contexts 配置文件 用于在编译和运行期间（如设备节点，服务套接字文件/init.rc 创建数据目录等）给文件打标签

seapp\_contexts 配置文件 用于标签应用进程和应用程序包目录

property\_contexts 配置文件 用于检查Android 的属性

mac\_permissions.xml 配置文件 是中间件MAC 策略

其中seapp\_contexts 配置文件和property\_contexts 配置文件是标准SELinux 源码中没有的文件。SEAndroid 系统源码作为Android 的一部分和系统源码一起编译并被加入到ramdisk.img 中。在ramdisk.img 中的安全库策略文件编译后为sepolicy.

#### 安全服务

用户空间的Security Server主要是用来保护用户空间资源的，以及用来操作内核空间对象的安全上下文的，它由应用程序安装服务PackageManagerService、应用程序安装守护进程installd、应用程序进程孵化器Zygote进程以及init进程组成。其中，PackageManagerService和installd负责创建App数据目录的安全上下文，Zygote进程负责创建App进程的安全上下文，而init进程负责控制系统属性的安全访问。

以下是对创建说明的进一步阐述：

应用程序安装服务PackageManagerService在启动的时候，会在/etc/security目录中mac\_permissions.xml文件，然后对它进行解析，得到App签名或者包名与seinfo的对应关系。当PackageManagerService安装App的时候，它就会根据其签名或者包名查找到对应的seinfo，并且将这个seinfo传递给另外一个守护进程installed。

守护进程installd负责创建App数据目录。 在创建App数据目录的时候，需要给它设置安全上下文，使得SEAndroid安全机制可以对它进行安全访问控制。Installd根据PackageManagerService传递过来的seinfo，并且调用libselinux库提供的selabel\_lookup函数到seapp\_contexts文件中查找到对应的Type。有了这个Type之后，installd就可以给正在安装的App的数据目录设置安全上下文了，这是通过调用libselinux库提供的lsetfilecon函数来实现的。

Zygote进程负责创建应用程序进程。 应用程序进程是SEAndroid安全机制中的主体，因此它们也需要设置安全上下文，这是由Zygote进程来设置的。组件管理服务ActivityManagerService在请求Zygote进程创建应用程序进程之前，会到PackageManagerService中去查询对应的seinfo，并且将这个seinfo传递到Zygote进程。于是，Zygote进程在fork一个应用程序进程之后，就会使用ActivityManagerService传递过来的seinfo，并且调用libselinux库提供的selabel\_lookup函数到seapp\_contexts文件中查找到对应的Domain。有了这个Domain之后，Zygote进程就可以给刚才创建的应用程序进程设置安全上下文了，这是通过调用libselinux库提供的lsetcon函数来实现的。

Init进程在启动的时候，会创建一块内存区域来维护系统中的属性以及Property服务。 这个Property服务通过socket提供接口给其它进程访问Android系统中的属性。其它进程通过socket来和Property服务通信时，Property服务可以获得它的安全上下文。有了这个安全上下文之后，Property服务就可以通过libselinux库提供的selabel\_lookup函数到property\_contexts去查找要访问的属性的安全上下文了。有了这两个安全上下文之后，Property服务就可以决定是否允许一个进程访问它所指定的属性了。

只有当进程和文件都关联安全上下文之后，SEAndroid安全策略才能发挥作用。也就是说，当一个进程试图访问一个文件时，SEAndroid会将进程和文件的安全上下文提取出来，根据安全策略规则，决定是否允许访问。

## SELinux 与Android 的结合

Android 的应用程序根据使用语言类型及是否调用链接库可以分为如下几类：

|  |  |
| --- | --- |
| 类别 | 说明 |
| 只使用Java 语言编写的程序 | 其默认的是通过Java 虚拟机即Dalvik 解析并执行，其权限控制方式有Android 自带的方式。除此外，SELinux 在底层也提供安全防护，但由于是通过虚拟机解析后的，默认其安全性能得到保障，所提供的策略保护是默认的类型，其安全使用系统默认策略。 |
| 基于C/C++语言的NDK 开发方式编写的程序 | 可以直接使用SELinux 的安全防护方式。应用程序访问资源方式是通过SELinux 回调函数，经过查询SELinux 策略库、判断后给予是否允许访问的结果。其与一般SELinux 的使用时候类似，故安全性能较高。 |
| Java 方式编写的程序中通过JNI 方式调用C/C++链接库 | Java 调用链接库\*.so 文件。该类型程序Java 语言执行时候适用正常的Android 安全策略方式，当其调用链接库时候，调用底层C/C++语言的库，此时适用SELinux 安全策略方式。 |

## 第5章 SEAndroid 的安全增强

MAC目前研究的Install-time MAC涉及的是应用软件APK中的AndroidManifest.xml文件和策略制定者编写的mac\_permissions.xml，Install-time MAC是一种比较严格的权限控制机制，当开发者在AndroidManifest.xml文件中申请的权限并没有包括在mac\_permissions.xml中，此时就没法进行应用程序的安装，因为AndroidManifest.xml中的权限必须是mac\_permissions.xml中对该应用程序制定策略的权限子集。

.SEAndroid 流程解析

在传统的权限配置层面上，SEAndroid区别于SELinux之外，针对Android的特殊性添加了property\_contexts和seapp\_contexts等文件。Install-time MAC中涉及到的文件主要有mac\_permissions.xml、seapp\_contexts和\*.te。

Install-time MAC存在的两个主线：

其中安装阶段对应于：APK->PMS扫描处理->生成package对象。

而运行阶段对应于：点击应用图标->ActivityManagerService->Zygote->fork产生进程。

## 第6章 SEAndroid 的权限检查

这里指的是MAC 中的Install-time MAC 的使用，SEAndroid 在Android 原有的permission 机制检查上加了新的一层过滤，主要体现在对策略库的进一步检查。在源码中是通过InstallPolicy 类进行检查的，InstallPolicy 类中存在解析及存储mac\_permissions.xml 文件的属性, 还有用来存储权限字符串的字段以及一个属性pkgPolicy (用来存储那些待检查的APK 中申明使用的权限字符串).

在InstallPolicy 类中存在一个方法determineInstallPolicyType 用来解析mac\_permissions.xml 文件，然后将应用对应的白名单和黑名单存放在相应的属性中，之后获得APK 中的权限属性。最后在该类中调用权限检查函数passedPolicyChecks进行权限字符串的匹配工作，该函数中就是使用迭代器得到权限字符串进行一一匹对，然后根据最后的结构来判断是否存在权限的乱用情况。

## 第7章 SEAndroid意义

SEAndroid安全机制的目的不是为了完全杜绝别人攻击我们的设备，而是为了保证我们的设备受到攻击时，受到的损害减少到最少的程度。例如，SEAndroid安全机制并不能完全阻止我们的设备被root，但是它能保证我们的设备被root之后，一些敏感的文件仍然是不可访问，这样就可以最大程度地保护我们的设备。这是因为只要程序是由人类来编写的，就或多或少地存在BUG，或者说漏洞，特别是复杂的程序，进而就会被黑客利用，并且成功地侵入到我们的系统中来。这是防不胜防的。当然，我们并不是说SEAndroid对阻止设备被侵入毫无用处，它在一定程度上还是能加大侵入的技术难度的。

## 第8章 SEAndroid 存在的问题

主要弊端: SEAndroid 系统须由开发者来预定义安全策略的弊端

SEAndroid 系统自身存在的最大的问题是沿袭了SELinux 固有的管理员操作方式，只有熟悉SELinux 系统的用户才有能力对系统的策略库进行定制，也就是从终端进行策略的修改，从而改变应用的安全上下文。这种操作方式很不符合面向普通消费者的移动终端操作系统的市场。

通常情况下，SELinux 下管理员需要自己编写应用的策略配置文件\*.te，然后进行编译得到policy.xx 文件，当内核启动的时候，将该策略数据库映射到内存中供安全服务器进行查询和控制。同时管理员需要根据不同的需要设置相应的策略规则，进行进程的类型切换，这些都是在以消费者为导向的Android 系统所不能适应的，因为普通的消费者不可能深入系统到这种程度，更不用说进行相应的安全控制了。因此灵活性和易用性有待加强。

# Android 8.0

策略分版本，system和vendor策略分离

8.0适应treble架构，有很大改变

原理可以学习老罗的博客再加上seandroid for treble，调试先切换成宽容模式确认是seandroid的问题，然后改策略。7.1及之前只需要编译和升级booting，8.0之后策略在system和vendor中，可以push进去调试。

# Te

无法打印内存信息

# 实战架构

# 远程签名整体架构

这个库最早出现在Github上, 是富士康的一个人开源的, 使用了Apache 2.0协议

<https://github.com/jasoncychueh/remote-sign>

# APK加固对比

## 加固策略

| **梆梆安全加固** | **爱加密** |
| --- | --- |
| 防反编译：对所有dex做方法抽取，加密；对dex1做整体加密； 防调试：对程序进场进行状态监控，防止调试 防二次打包：对程序运行进行判断，防止二次打包 So文件资源文件：raw/、r数据文件：shared\_prefs/\*.xml | VMP加密nager So文件：  资源文件：res/r  数据文件：shared\_prefs/\*.xml |

## 性能影响

首次启动速度基本上无影响

## 黑屏问题

问题描述：分dex，首次安装的黑屏，之后打开都没问题。

| **梆梆安全加固** | **爱加密** |
| --- | --- |
| 在加固的APP上黑屏时间更长（梆梆已处理） | 每次打开都会 |

## 加固调整后是否保护到位

### 防反编译

| **梆梆安全加固** | **爱加密** |
| --- | --- |
| 1. 加固前保留在maindex中的类都没有dump出来。（例如：\*\*） 2. 虽然其他的类都dump出来了，但是关键的函数实现都被抽离了，只剩一个空方法{}，如果有返回值的就{return 0}。 3. 对于虚拟指令集，梆梆那边说目前知道影响的有影响有扫码、画图和大量计算，都在10ms的量级。 | 1. 加固后反编译出来的dex只剩爱加密自己的dex。 2. dex2jar得到的jar包也只有爱加密相关的类 3. 虚拟指令集，暂时没有询问爱加密相关信息，估计跟梆梆影响差不多 |

### 防调试

| **梆梆安全加固** | **爱加密** |
| --- | --- |
| 暂无 | 暂无 |

### 防二次打包

| **梆梆安全加固** | **爱加密** |
| --- | --- |
| 用自己生成的签名给二次打包后的APK签名，安装后打开就闪退 | 用自己生成的签名给二次打包后的APK签名，安装后打开一直卡在启动页 |

### 数据文件加固

看**升级到加固版本，对原来用户数据是否有影响**部分

## 升级到加固版本，对原来用户数据是否有影响

## 对crash的定位是否有影响

# VMProtect

# VMProtect技术原理

VMP主要是将原来的可执行代码转换为梆梆自定义的虚拟机指令，这些虚拟机指令只能由梆梆自定义的虚拟机解释器解释并执行。

# VMProtect的使用

@Retention(RetentionPolicy.CLASS)

@Target(ElementType.METHOD)

public @interface ProtectMeVmpMethod {

}

@ProtectMeVmpMethod

private void VMPTest()

{

  Log.i("TAG", "Hello VMProtect");

}

# VMProtect的过滤规则

* 过滤带有abstract修饰符的类或在interface、enum类型中的方法
* 带有abstract、native、synthetic、init、bridge修饰符的方法
* 方法名为access$跳过
* 子类和父类有一个相同的变量，不一定是子类和父类也有可能是父类的父类，这个函数用到了这个变量，这个函数做vmp的话就会有问题，所以是自动给过滤掉了
* 如果父类和子类有一个共同的方法名字a，分为两种情况，一个父类的a是private的＋子类的a是public的。还有一种父类的a是private的＋子类的的a是protected的。这两种情况都过滤。
* 这个方法名字是run，然后这个方法所在的类implement 这个java/lang/Runnable类
* com.bangcle开头的类不做vmp
* 方法名字是getAnnotation的不做
* 引用了另外类的静态变量（如果虚拟机自己优化了，就会找不到所对应的field id）

# 标准

| **验收功能** | **验收方式** | **说明** |
| --- | --- | --- |
| 防逆向保护(Anti-RE) | apktools等反编译工具无法对dex进行反编译操作 | 以加密代码的方式阻止编译，从而防止被窃取代码和创意 |
| 防篡改保护(Anti-tamper) | 安装包内容或签名被替换后，无法正常使用 | 通过对APP的完整性保护，防止APP被篡改或盗版 |
| 反调试保护(Anti-debug) | IDA、exposed等调试注入工具，无法对app进行动态调试注入操作 | 阻止应用运行中被动态注入，防止被外挂、木马偷窃账号密码，修改交易金额等 |
| so文件保护(so-protected) | 无法通过IDA等静态分析工具查看汇编代码，分析代码逻辑 | 对so进行binary级别的加密包，防止C代码的泄露 |
| 资源文件加密保护(Resource- Encryption) | 无法查看修改资源文件 | 更底层、跨文件格式的数据加密，防止应用数据被窃取 |
| 存储数据加密保护(Storage Encryption) | 数据在存储时做加密处理，用户拿到文件也无法读取查看内容 | 更底层、跨文件格式的数据加密，防止应用数据被窃取 |
| 安装包大小 | 安装包大小不能有明显增幅，增幅不能超过原大小的8% |  |
| 兼容性 | 在主流兼容性平台，兼容性测试，加固前和加固后，兼容性测试通过率差异不能大于0.5% |  |
| app崩溃率 | 使用梆梆加固后，app崩溃率不能有明显增幅，增幅不能大于0.1% |  |

有3层保护。

第一层保护是dex加壳，把我们的maindex替换成他们自己dex，然后他们再用classLoader来加载我们的dex。

第二层保护是类跟方法的实现抽离，加载的时候再拼接在一起。

第三层保护是虚拟指令集（相关方法最终从虚拟指令集里调用，这种加密的映射关系只有梆梆知道）。

1. 反编译apktool d DJI-170725-292\_gpPhone.apk
2. 二次编译apktool b DJI-170725-292\_gpPhone
3. 用自己生成的签名给二次编译的APK签名jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore /Users/jackson.yuan/Documents/Android/appkey.jks /Users/jackson.yuan/Downloads/safeTest/DJI-170725-292\_gpPhone/dist/DJI-170725-292\_gpPhone.apk appkey
4. 安装签名后的APK，能正常打开。

### so文件加固

1. 把加固前的so的符号表dump出来arm-linux-androideabi-objdump -D D /lib/armeabi-v7a/$lib\_name.so > $lib\_name.org.txt
2. 把加固后的so的符号表dump出来arm-linux-androideabi-objdump -D al\_crashtest\_1501662554.6\_sec/lib/armeabi-v7a/$lib\_name.so > $lib\_name.safe.txt

情况如下图所示。（org表示加固前，safe表示加固后）

# 核心概念

### 防内存窃取

防止通过gdb、gcore，从内存中截取dex文件，获取代码片段，从而反编译还原APK进行不法操作。

### 防动态跟踪

防止通过ptrace调试进程，跟踪、拦截、修改正在运行的应用，进行动态注入，保护程序运行安全。

### 防逆向分析

防止通过APKTool、IDA Pro等反编译工具破解DEX文件，从而获取APK源代码，保护代码层安全。

### 防恶意篡改

校验APK完整性，自动终止运行被篡改的APK，二次打包后应用都无法使用，杜绝盗版应用的出现。

### 存储数据加密保护

更底层，跨文件格式的数据加密，防止应用数据被窃取